Model Checking Generic Container Implementations [⋆]

Matthew B. Dwyer and Corina S. Păsăreanu

Kansas State University,
Department of Computing and Information Sciences,
Manhattan KS 66506, USA
{dwyer,pcorina}@cis.ksu.edu

Abstract. Model checking techniques have been successfully applied to the verification of correctness properties of complex hardware systems and communication protocols. This success has fueled the application of these techniques to software systems. To date, those efforts have been targeted at concurrent software whose complexity lies, primarily, in the large number of possible execution orderings of asynchronously executing program actions. In this paper, we apply existing model checking techniques to parameterizable implementations of container data structures. In contrast to most of the concurrent systems that have been studied in the model checking literature, the complexity of these implementations lies in their data structures and algorithms. We report our experiences model checking specifications of correctness properties of queue, stack and priority queue data structures implemented in Ada.

Keywords : Model checking, temporal logic, assume-guarantee reasoning, generic containers

1 Introduction

The past three decades have seen the development of a large number of formal methods for specifying the intended computational results of a software component or system. Clearly, the practical utility of a formal method must be assessed with respect to its ability to describe and support reasoning about the correctness of realistic software systems and a number of efforts in this area have been made (e.g., [1, 7]). Short of such case studies, a basic test of a formal method is its ability to address certain standard examples; one class of standard examples are container data types (e.g., stacks, queues, sets).

More recently, in the past decade, a number of formal methods researchers have focused on a class of specification formalisms and automated specification checking techniques referred to as *finite-state verification* (FSV) approaches. Rather than focusing on specification of the correct results of a computation,

[⋆] Supported in part by NSF under grants CCR-9703094 and CCR-9708184, by NSF and DARPA under grant CCR-9633388, and by NASA under grant NAG-02-1209.

FSV methods specify correctness properties related to, perhaps internal, system behaviors. Such behaviors might include, for example, correctness of component coordination, safety of data access, and guarantee of progress in performing a computation. These techniques, in particular model checking, have enjoyed success in finding defects in and verifying properties of real hardware systems (e.g., [4, 9, 20]). Attempts have been made to capitalize on the success of model checking by applying it to selected software domains (e.g., communication protocols [16] and control systems [17]). Control software, typically, manipulate small amounts of data that influence system behavior. More general software systems, however, may manipulate large amounts of data and use that data to control system behavior. It remains to be seen whether FSV approaches will be generally effective for such data-intensive software.

In this paper, we assess a representative of the FSV approaches, namely linear temporal logic (LTL) [19] model checking using SPIN [16], in terms of its ability to support reasoning about the standard container implementations on which formal methods are often judged. Model checking is a technique for sound and complete reasoning about finite-state transition systems. To enable model checking of software, however, completeness is sacrificed for tractability and because software systems are, in general, not finite state. Our methodology [10] uses correctness preserving abstractions of system components to render model checking sound. We believe that such model checking complements traditional formal methods and testing as a software quality assurance technique. Like testing, model checking is an automated technique. Thus, unlike most traditional formal methods which require some user interaction, model checking can be applied by practitioners with little knowledge of proof construction strategies. Like traditional formal methods, sound model checking when successful provides a guarantee that the system satisfies the property being checked; testing is unable to guarantee the absence of errors with respect to a given property. While a failed test is a sure indication of a defect in the system, it provides no guidance to help locate the cause of the defect. In contrast, both proof-based methods and model checking produce a counter-example, which traces an example system execution that violates the property of interest. Analysis of such counter-examples often leads directly to the cause of the erroneous system behavior. Given these qualitative differences between model checking and other methods, our goal is not to compare classes of software validation approaches, but to assess the extent to which model checking can be applied to validate and detect defects in existing container implementations.

We use tools to automate the process of translating a program written in Ada to a safe finite-state model rendered in the input format of SPIN. For this reason, we selected implementations of generic queue, stack and priority queue data types written in Ada. We specified a variety of correctness properties of these abstractions in LTL and checked them using SPIN. Our results are encouraging and suggest that model checking tools can be effectively applied to detect defects in such implementations. We report these results and describe the process we followed to carry out this study.

In the following section we discuss relevant background material. The construction of finite-state models of the container implementations is outlined in Section 3. Section 4 describes the container implementations and Section 5 describes the correctness properties that we analyze. We then present, in Section 6, the results of our study. Section 7 describes this study's limitations and suggests directions for further study and Section 8 concludes.

2 Background and Related Work

In this section, we give a brief overview of temporal logic and model checking. We focus on the differences between this approach to system specification and verification and more traditional formal methods.

2.1 Traditional Formal Methods

Traditionally, formal methods have been designed to support the precise description of the intended results of a computation. A specification is written such that a conforming implementation is guaranteed to produce the correct result; a specification defines a sufficient condition for correctness. There are a wealth of formal methods including model-based (e.g., Z [8]), algebraic (e.g., Larch [18]), and trace-based (e.g., [15]) methods. While these methods differ in their formal underpinnings, each is expressive enough to describe computations over unbounded data domains, such as the naturals. Thus, each is capable of specifying container data structures. In fact, specifications for stacks and queues are often given as examples to illustrate a method [8, 15, 18].

Trace specifications [15] are particularly relevant to the work described in this paper. These are abstract specifications of the observable interface behavior of a software component. A trace specification defines the legal sequences of calls to operations in a component's interface and the computational effects of such sequences. Hoffman and Snodgrass distinguish three different components of such a trace specification: legality, equality, and value. The specifications we describe in Section 5 state necessary conditions for such trace components, concentrating primarily on the legality of a sequence of calls. Like the stack and queue specifications presented in [15], our stack and queue specifications are nearly identical, differing only in the names of operations and element ordering. We note that our specifications model invocation and return from a procedure as independent atomic actions, whereas traces model them as a single action. We do this to enable specification of properties of concurrent systems where multiple calls to a single procedure may co-execute.

2.2 Temporal Logic

For model checking, specifications are written in a propositional temporal logic. These logics are less expressive than the formalisms used in traditional formal methods (e.g., they cannot express properties of unbounded value sets). For this reason model checking focuses on specifications of necessary conditions for correctness, rather than complete specification of a system's computational effects.

We use linear temporal logic in our work because it supports unit-level model checking, through filter-based analysis [11, 12], and it is supported by a robust tool, SPIN [16]. In LTL a pattern of states is defined that characterizes all possible behaviors of the finite-state system. We describe LTL operators using SPIN's ASCII notation. LTL is a propositional logic with the standard connectives `&&`, `||`, `->`, and `!`. It includes three temporal operators: `<>`p says p holds at some point in the future, `[]`p says p holds at all points in the future, and the binary p`U`q operator says that p holds at all points up to the first point where q holds. An example LTL specification for the response property "all calls to procedure P are followed by a return from P" is `[](call_P -> <>return_P)`.

2.3 Model Checking Software

In model checking, one describes software as a finite-state transition system, specifies properties with a temporal logic formula, and checks, exhaustively, that the sequences of transition system states satisfy the formula. In principle, model checking can be applied to any finite-state system. For software one cannot render a finite-state system that exactly models the software's behavior, since, in general, software will not be finite-state. Even for finite-state software the size of a precise finite-state model will, in general, be exponential in the number of independent components (i.e., variables and threads of control). For these reasons, we use abstracted finite-state system models that reflect the execution behavior of the software as precisely as possible while enabling tractable analysis.

Existing model checkers, such as SPIN, do not accept Ada source code. In fact, the semantic gap between Ada and a model checker input language, such as SPIN's Promela, is significant. To bridge this gap, and achieve tractable model checking, we perform an "abstract compilation" of Ada source code to a target program in Promela. This compilation is guided by the LTL formula to be checked with the result that target program is sound with respect to model checking of that formula. In this setting, positive model checks results for a specification imply conformance of the original Ada implementation to that specification. A failed model check result is a trace of the target program that violates the specified property called a counter-example. Analysis of the counter-example indicates one of two situations: (*i*) the implementation is defective with respect to the specified property, or (*ii*) the abstractions used in the compilation of the Promela program are too imprecise for checking the specified property. In the second case, the counter-example provides guidance as to which abstractions must be strengthened to enable a successful model check or the detection of a defect. This compilation process is discussed in more detail in Section 3.

2.4 Filter-based Analysis

It is common in software validation and verification to reason about parts of an implementation in isolation (e.g., unit testing). For unit-level software model

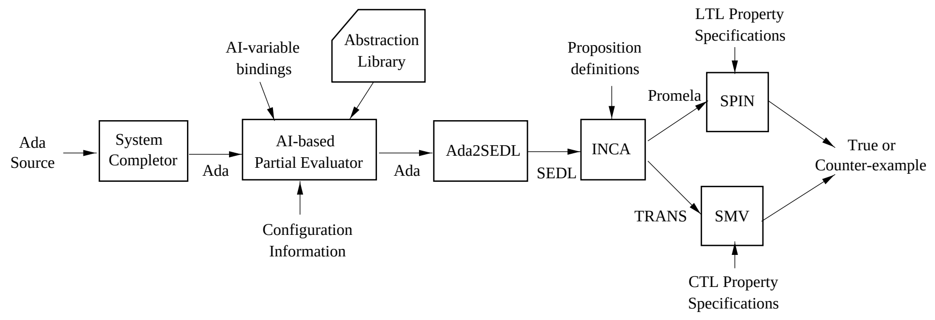

Fig. 1. Model Construction Process

checking, we adopt the assume-guarantee paradigm [22] in which a system description consists of two parts: a model of guaranteed behavior of the software unit and a model of the assumed behavior of the environment in which that unit will execute. In the work described in this paper, we define an Ada driver program that approximates the behavior of all possible contexts in which a software unit may be invoked; we discuss such driver programs in the next section. Model checking of properties of the software unit combined with the driver is sound with respect to any usage of the unit.

Many software components make assumptions about the context in which they will be used (e.g., that a called routine will eventually return). Such assumptions can be incorporated into LTL model checking using SPIN [11,12] as follows: given a property P and filters $F_1, F_2, \ldots F_n$ that encode assumptions about the environment, we model check the combined formula $(F_1 \&\& F_2 \&\& \ldots \&\& F_n) \rightarrow P$. We refer to the individual F_i as *filters* and to the combined formula as a *filter-formula*.

One concern with this method is that users may make invalid assumptions. We address this in two ways. First, we check that the assumptions are not inconsistent by checking that there exists an execution that satisfies the filters; with SPIN this is done by attempting to falsify a never claim for the conjoined filters. Second, when a user wishes to use the fact that a given software component satisfies P in the analysis of the larger system we model check $(F_1 \&\& F_2 \&\& \ldots \&\& F_n)$ on the sub-systems that use the component; this verifies that the context satisfies the assumptions.

3 Model Construction

In this section, we describe the process of compiling programs to descriptions suitable for model checking. We apply the methodology described in [12] which is supported by the toolset illustrated in Figure 1. Incompletely defined Ada source code is fed to a component which constructs a driver program to complete its definition. This completed program is then systematically abstracted and simplified using abstract interpretation and partial evaluation techniques. The resulting program is converted to the input language, called SEDL, of the INCA

```
procedure Driver() is
 choice : Integer;
 theObject_Type : Object_Type;
begin
 loop
  case choice is
   when 1 => Insert(theObject_Type);
   when 2 => theObject_Type := Remove;
   when 3 => null;
   when others => exit;
  end case;
 end loop;
end stub;
```

Fig. 2. Driver Source Code

toolset [2, 5]. INCA accepts definitions of the states and events that form the propositions used in specifications and embeds those propositions into the model checker input. A tutorial on the use of this toolset is available [13].

3.1 Completing Partial Systems

A *complete* Ada program is constructed by defining a driver program and stub routines for referenced components that are external to a given partial Ada program. This is similar to a unit-testing approach, except that the stubs and driver are capable of calling the component's public operations in any possible sequence; this simulates any possible usage scenario that may arise for the component. Together, the stubs, driver and component under analysis form a complete Ada application.

To simplify the discussion, we assume that a partial system is encapsulated in a package with public procedures that can be called from outside the package and that we are only interested in the behavior of the partial system (not the environment). A driver program is generated that will execute all possible sequences of public procedure calls. The driver program defines local variables to hold parameter values. Figure 2 illustrates a driver for a partial system with an `Insert` procedure and a `Remove` function. Note that the `choice` and `theObject_Type` variables will be subsequently abstracted; `choice` will be abstracted to model nondeterministic choice of the case statement alternatives. Stub routines are defined in a similar way.

Ultimately, in order to generate a finite-state model from source code all layering of the container implementations must be removed (e.g., the calls to `Insert` and `Remove` in `Driver` must be inlined). The INCA tools perform inlining of non-recursive procedure calls[1]. The tools are not able to perform generic instantiations, so for our examples this was done by hand.

[1] A limitation of our current approach is that it does not treat recursive procedures.

3.2 Incorporating Abstractions

The methodology then proceeds by binding abstract interpretations [6] to selected program variables. A variety of sound abstract interpretations have been defined for different data types. Currently, a set of heuristics are applied to select the variables that are to be abstracted and the specific abstract interpretations to be used for each such variable [12].

The machinery of partial evaluation [14] is used to propagate these abstract variable definitions throughout the program's definition. In this way, the state space of the finite-state transition system constructed for the program can be safely reduced to a size that enables efficient model checking. For the container implementations we studied, almost all variables were defined over discrete ranges (e.g., sub-range types for array indices, booleans); for those variables no abstractions were applied.

The container implementations are parameterized by the type of contained data elements. The properties we check of those implementations, which are described in Section 5, do not specify behavior in terms of data element values, rather they refer to the identity of the data elements. Some properties are related to the ordering of two data elements. To construct a safe model that supports reasoning about such properties we use the *2-ordered data* abstraction [12]. This defines data elements in terms of their identity where two elements (named `d1,d2`) are distinguished from all others (named `ot`). For example, the `Object_Type` in Figure 2 is replaced by the enumerated type `(d1,d2,ot)` which implements this abstraction. Since container implementations only assign data elements and compare them based on identity, the 2-ordered abstraction allows us to treat containers as data-independent systems [23] and prove order related specifications under the assumption that `d1` and `d2` are input to the container at most once. These proofs generalize to any pair of user-defined data elements.

3.3 Specializing Source Code

Partial evaluation can also produce a specialized version of the software system when supplied with information by the user. For systems with dynamically sized data structures it is often the case that specialization is required to place an upper-bound on the number of dynamic objects to allow a finite-state model to be constructed. While this reduces the generality of any property proven of that model, in practice it preserves much of the effectiveness of model checking for defect detection.

The container implementations we consider in our study accept a single constructor parameter that pre-allocates storage for holding all data. We constructed drivers which invoked the constructor with small size values (e.g., 3) and checked properties of the resulting models.

4 The Containers

The Scranton Generic Data Structure Suite [3] is a publicly available collection of over 100 Ada packages that implement a number of variations of list, queue, stack, heap, priority queue, binary and n-ary tree data structures and algorithms on those structures. We selected three specific data structure implementations from version 4.02 of this collection as the subject of our study: `queue_pt_pt`, `stack_pt_pt`, and `priority_queue_lpt_lpt`[2]. The queue and stack are implemented using support packages which provide list and iterator implementations. The priority queue is implemented using underlying heap and iterator implementations. The stack, queue, and priority queue implementations are generic packages requiring a type parameter for the data to be stored in the container. The priority queue also requires a type for the priority of a contained data item as well as an ordering operation, `<`, on the priority type. The queue and stack implementations both allocate a fixed-size array of the appropriate object type for storage of data. The priority queue is implemented on top of an array-based heap implementation; three levels of generic package instantiation are required to reveal the structure of the concrete implementation. The queue and stack container types have associated iterator packages which support iteration in container-order (`Top_Down` iteration) or in reverse-container-order (`Bottom_Up` iteration). Iterators require a user defined call-back routine (`Process`) that is invoked for each datum stored in the container.

To denote the different models, we use `s`, `q`, `p` for the stack, queue and priority queue packages in isolation. The queue with forward and backward iterators required different models `qt` and `qb`, respectively. The stack with iterator required a single model `si`. Finally, each of these models can be scaled for a fixed maximum number of contained elements, e.g., `s(3)` is a stack of at most 3 elements.

5 The Properties

As discussed in Section 2, we cannot hope to specify the complete behavior of the container abstractions in LTL. Instead we focus on several crucial correctness properties of these abstractions. We group these in three areas: containment, order, and observation. Containment properties are related to whether a container implementation has knowledge of data items that have been put into it and not yet taken out of it. Order properties are related to the order in which data items can be removed from or observed in a container (e.g., that priority ordering, via `<`, is observed). Observation properties are related to the ability of operators to distinguish relevant features of the state of the container (e.g, whether a container is empty or not).

We wrote specifications in each of these categories for the three container implementations. In writing specifications one must select a level of abstraction at which to describe system behavior. For our study, we chose the package interfaces for the containers. Specifically, for our LTL specifications we define propositions that describe calls to container operations with specific input parameter values and returns from container operations with specific output parameters. We illustrate the syntax for the names of these propositions by way of example. For

[2] The naming scheme indicates whether the type of the contained data and the instantiated container itself is private, `pt`, or limited private, `lpt`.

the following procedure declaration:

```
 procedure And(x,y : in Boolean; r : out Boolean)
```

we would have a number of propositions, including:

`call_And`	representing any call to `And`
`return_And`	representing any return from `And`
`call_And(true,false)`	representing any call to `And` with `x=true` and `y=false`
`call_And(,false)`	representing any call to `And` with `y=false`
`return_And(false)`	representing any return from `And` with `r=false`

Function return values are considered to be the last output parameter. With these propositions we can state that "Calling `And` with `true` parameters should return `true`" as

```
 [](call_And(true,true) -> !return_And U return_And(true))
```

This states that any call to `And` with `true` parameters must be followed by a return from `And` with the value `true` and that no other return from `And` can intervene between the the designated call and return. Our writing of specifications in LTL was greatly simplified by use of a specification patterns system [10]; the above specification is an instance of the *global constrained response* pattern.

Figure 3 illustrates a sampling of the specifications that were checked in our study; space limitations prohibit showing all of the specifications. There are 5 basic properties specified. Each of these properties has a slightly different intended semantics depending on the container being checked; we use `q`, `s`, and `p` in the property names to indicate queue, stack and priority queue versions, respectively. The containment specifications, (1), vary depending on whether forward or backward iteration is performed and on the name of the container insert(remove) operation `Enqueue` or `Push`(`Dequeue` or `Pop`) (e.g., (1fq) and (1bs)). For observation specifications, (2-3), only the names of the insert(remove) operations vary. For ordering specifications, (4-5), the modifications are more complex. The order of returned data for the queue and stack are reversed (e.g., (5q) (not shown) and (5s)). The order of dequeued data for the priority queue depends only on the priority value of a datum, thus, we specify the same return sequence regardless of insertion order (e.g., (5p21) and (5p12) (not shown)). There is an assumption about data items `d1` and `d2` that is important for reasoning about the order related properties; this assumption is discussed in the next section, In total there are 18 LTL specifications for variations of the 5 basic properties.

6 Analysis Results

A selection of the specifications we checked was given in Section 5. All model checks were performed using SPIN, version 3.09, on a SUN ULTRA5 with a 270Mhz UltraSparc IIi and 128Meg of RAM (machine 270) or on a SUN Enterprise 4000 with a 168Mhz UltraSparc II and 512Meg of RAM (machine 168). Figure 4 gives the data for each of the model checking runs; the transition system model used for the run is given[3]. We report the elapsed time for running

[3] Detailed description of the transition systems is published as a case-study at `http://www.cis.ksu.edu/santos`

(1fq) If a datum is enqueued, and not dequeued, then forward iteration will invoke the iterator call-back with that datum, unless the iteration is terminated.

```
[](call_Enqueue(d1) && ((!return_Dequeue(d1)) U call_Top_Down) ->
   <>(call_Top_Down && <>(call_Process(d1) || return_Process(,false))))
```

(1bs) If a datum is pushed, and not popped, then forward iteration will invoke the iterator call-back with that datum, unless the iteration is terminated.

```
[](call_Push(d1) && ((!return_Pop(d1)) U call_Bottom_Up) ->
   <>(call_Bottom_Up && <>(call_Process(d1) || return_Process(,false))))
```

(2q) An enqueue without a subsequent dequeue can only be followed by a call to empty returning false.

```
[](call_Enqueue && (!return_Dequeue U call_Empty) ->
   <>(call_Empty && <>return_Empty(false)))
```

(3p) Between an enqueue of a datum and a call to empty returning true there must be a dequeue returning that datum.

```
[]((call_Enqueue(d1) && <> return_Empty(true)) ->
   (!return_Empty(true) U (return_Dequeue(d1))))
```

(4fq) If a pair of data are enqueued, and not dequeued, then forward iteration will invoke the iterator call-back with the first datum then the second, unless the iteration is terminated.

```
[]((call_Enqueue(d1) && ((!return_Dequeue(d1)) U (call_Enqueue(d2) &&
   ((!return_Dequeue(d1) && !return_Dequeue(d2) U call_Top_Down)))) ->
   <>(call_Top_Down && <>((call_Process(d1) && <>call_Process(d2)) ||
                            return_Process(,false))))
```

(5s) If a pair of data are pushed then they must be popped in reverse order, if they are popped.

```
[]((call_Push(d1) && (!return_Pop(d1) U call_Push(d2))) ->
   (!return_Pop(d1) U (return_Pop(d2) || [](!return_Pop(d1)))))
```

(5p21) If a pair of data are enqueued in reverse-priority order, then they must dequeued in priority order (`Priority(d1)>Priority(d2)`).

```
[]((call_Enqueue(d2) && ((!return_Dequeue(d2)) U (call_Enqueue(d1)))) ->
   (!return_Dequeue(d2) U (return_Dequeue(d1) || [](!return_Dequeue(d2)))))
```

Fig. 3. LTL Specifications

SPIN to convert LTL to the SPIN input format, to compile the Promela into a model checker, and to execute that model checker. The model construction tools were run on an AlphaStation 200 4/233 with 128Meg of RAM. The longest time taken to convert completed Ada to SEDL was for the model of a priority queue of size 3; it took 26.7 seconds. Generating Promela from the SEDL can vary due to differences in the predicate definitions required for different properties. The longest time taken for this step was for the model of the queue of size 2; it took 129.3 seconds.

Property	*Time*	*Result*	*Model*	*Machine*
(1fq)	0.2, 54:12.9, 7.6	false	qt(2)	270
(1fq_f)	0.9, 1:43:36.2, 9.4	true	qt(2)	270
(1bq)	0.1, 54:39.9, 7.6	false	qb(2)	270
(1bq_f)	1.0, 1:27:39.9, 13.9	true	qb(2)	270
(1fs)	0.2, 1:12.6, 0.1	false	si(2)	270
(1fs_f)	0.9, 2:01.6, 0.2	true	si(2)	270
(1bs)	0.2, 1:12.2, 0.2	false	si(2)	270
(1bs_f)	0.9, 1:53.1, 0.2	true	si(2)	270
(2q)	0.1, 26.1, 0.2	true	q(2)	270
(2s)	0.1, 23.6, 0.2	true	s(3)	270
(2p)	0.1, 21.2, 0.1	true	p(3)	270
(3q)	0.1, 14.0, 0.1	true	q(2)	270
(3s)	0.1, 14.1, 0.1	true	s(3)	270
(3p)	0.1, 10.4, 0.1	true	p(3)	270
(4fq)	6.9, 7:57:43.9, 12.6	false	qt(2)	168
(4fq_f)	1:59.5,11:41:56.9, 37.0	true	qt(2)	168
(4bq)	8.5, 7:44:51.6, 14.4	false	qb(2)	168
(4bq_f)	1:27.8, 9:40:58.7, 27.5	true	qb(2)	168
(4fs)	6.7, 7:08.5, 0.2	false	si(2)	270
(4fs_f)	1:24.6, 10:45.2, 1.1	true	si(2)	270
(4bs)	8.8, 7:12.1, 0.2	false	si(2)	270
(4bs_f)	2:00.8, 10:35.6, 0.7	true	si(2)	270
(5q)	0.1, 19.0, 0.1	false	q(2)	270
(5q_f)	11:57.5, 15:44.2, 2.6	true	q(2)	270
(5s)	0.2, 25.5, 0.1	false	s(3)	270
(5s_f)	9:34.9, 13:03.4, 3.6	true	s(3)	270
(5p12)	0.1, 11.9, 0.1	true	p(3)	270
(5p21)	0.2, 11.6, 0.1	true	p(3)	270

Fig. 4. Performance Data

Our container models make no assumptions about the way that the container operations are invoked (e.g., number of times, order of invocation). The iterator call-back `Process` is assumed to be data-independent. To boost precision in checking certain properties, we code assumptions about the required behavior of the driver or stub as a filter and then model checked the filter-formulae (denoted by "_f" in the table).

For example, analysis of the counter example provided by SPIN for specification (1fq) showed that the result is false because it is possible for the stub for `Process` to never return. Enforcing the reasonable assumption that the iterator call-back always returns, yields the filter-formula, (1fq_f):

```
[](call_Process -> <>return_Process) ->
[]((call_Enqueue(d1) && (!return_Dequeue(d1) U call_Top_Down)) ->
   <>(call_Top_Down && <>(call_Process(d1) || return_Process(,false))))
```

The same filter was used for other properties of type (1) and (4).

The use of filters in properties of type (5) was required since the 2-ordered data AI incorporated in the model is only guaranteed to be safe under the assumption of a single insertion of each data item into the container. For example, the filter-formula (5s_f) is:

```
([](return_Push(d1) -> [](!call_Push(d1))) &&
 [](return_Push(d2) -> [](!call_Push(d2)))) ->
[]((call_Push(d1) && (!return_Pop(d1) U call_Push(d2))) ->
   (!return_Pop(d1) U (return_Pop(d2) || [](!return_Pop(d1)))))
```

These results are consistent with previous work on filter-based analysis [12, 21]. When filters are required they are relatively few and simple. For the most part (ignoring SPIN compile times), this study shows that the total time required to model check properties of container implementations is on the order of a few minutes. We discuss the compile-time issue in the next section.

7 Discussion and Future Work

While we believe that our results indicate the potential for model checking to be an effective quality assurance technique for a broad class of software systems, there are a number of clear limitations to our study.

7.1 Scaling Containers

We believe that the results of the previous section bode well for using model checking techniques for reasoning about software systems, but, there remain significant questions about its ability to scale to large systems. We used containers of size 2 and 3 for the bulk of our model checks. These are the smallest sizes that one would consider using for reasoning about order properties. When we increased container size to 4 for the priority queue, model generation and check times increased by a factor of 3. For stack and queue containers of size 4 the model generation tools ran for 10 hours before we stopped them. As one would expect there is a very rapid growth in the size of the state space as container size increases. Since the model generation tools expand parts of this state space they require significant amounts of memory. We believe that memory limitations were one cause of the significant slowdown in generation time with increasing size. Future experiments with large-memory machines will help address this question.

Our model generation tools (i.e. INCA) were designed for reasoning about synchronization properties of concurrent systems [5]. Such systems typically have little data that is used to control the pattern of inter-process synchronization. Given these requirements it was a reasonable design decision to encode all data local to a single process into the control flow of that process. Unfortunately, for data intensive systems like containers, this causes an enormous expansion in the program's control flow. This is why the Promela compile-times in Figure 4 are so large (nearly 12 hours for (4fq_f)). It is important to understand that this is an artifact of the model construction process and not an inherent limitation of SPIN. To illustrate this, we checked the queue properties of type (4) on a model that was hand translated from Ada to Promela (converting Ada variables into Promela variables). The effect was dramatic, for (4fq_f) Promela compile-time was 4.1 seconds and model check time dropped to 22.3 seconds. Future versions of model generation tools will need to define their mappings with such performance issues in mind.

A variety of different model checking techniques have been developed. Given the data-intensive nature of container implementations we wondered whether a different model checking technique might work better. In particular, whether SMV [20] and its use of OBDD-based encodings of transition systems might be effective in compactly representing the data state-space of the container models. INCA generates very efficient input models for SMV. We re-ran all of the priority queue property model checks using SMV 2.5 on machine `270`. Model generation time was 2.2 seconds and for the (5p12) and (5p21) properties model check time was 0.7 seconds. It appears that SMV may be more effective than SPIN for checking properties of this kind of system. Unfortunately, SMV's specification language cannot easily incorporate filter-formula and that is a significant limitation for assumption-based validation of partial software systems.

7.2 Dynamism in Implementations

Many container implementations do not pre-allocate storage for their contents, rather they dynamically allocate that storage as needed. To check properties of such implementations we need to incorporate a safe abstraction of allocation and reclamation of heap storage. We are investigating the use of a scalable bounded heap abstraction that allows allocation and deallocation of data and tracks the state of the heap. Whether the cost, in the size of the finite-state system model and consequent model check time, is prohibitive is the subject of future empirical study.

7.3 Defect Detection

While model checking is capable of verifying properties of software, its main benefit may be as a fault-detection technique. Fortunately, for many systems faults are exhibited in small system sizes where application of model checking is most cost-effective.

To illustrate this we seeded a fault in the priority queue implementation. In the `Insert` procedure from the heap package, presented in Figure 5, we deleted the `not` from the test of the `while` loop. We then re-ran the model checker for (5p21) and detected the defect in essentially the same time as reported in Figure 4. SPIN produced a counter-example that gives the changes in the values of the propositions along a path through the system on which the property does not hold. The simulation output generated by SPIN from the counter example with only the boolean variables for the predicates that appear in property specification (5p21) is given in Figure 6. It is easy to see from this counter-example, that the data is not dequeued in priority order (i.e., `Priority(d1)` > `Priority(d2)`).

```
procedure Insert (Heap : in out Heap_Type; Object: in out Object_Type) is
 Parent: natural := (Heap.Size + 1) / 2 ;
 Child : natural := Heap.Size + 1 ;
 begin
   if (Heap.Size = Heap.Max_Size) then
    raise Heap_Overflow ;
   else
    Heap.Size := Heap.Size + 1 ;
    Initialize (Heap.Data.all(Heap.Size));
    while (Parent>0) and then not ((Heap.Data.all(Parent)>=Object)) loop
      Swap (Heap.Data.all(Parent), Heap.Data.all (Child)) ;
      Child := Parent ;
      Parent:= Parent / 2 ;
    end loop;
    Swap (Object, Heap.Data.all(Child) ) ;
   end if;
 end Insert ;
```

Fig. 5. Insert procedure

```
53: proc 2 (driver_task) line 150 "pan_in" (state 143) [callEnqd2=1]
66: proc 2 (driver_task) line 165 "pan_in" (state 159) [callEnqd2=0]
119: proc 2 (driver_task) line 8179 "pan_in" (state 8916) [callEnqd1=1]
132: proc 2 (driver_task) line 8194 "pan_in" (state 8932) [callEnqd1=0]
191: proc 2 (driver_task) line 2227 "pan_in" (state 2402) [returnDeqd2=1]
```

Fig. 6. Reduced SPIN counter-example

This faulty result either indicates a defect in the implementation or some imprecision in the finite-state model. We use only one data abstraction (the *2-ordered data* abstraction) which is safe under the assumption that d1 and d2 are input to the container at most once [12]. However, in the counter-example, d1 and d2 are enqueued only once and thus, the assumption is not violated. Having eliminated the possibility of an imprecise abstraction the only conclusion is that the implementation has a defect.

7.4 Research Issues

Our study revealed several interesting research issues that we intend to explore in future work.

The properties we considered in this study are *necessary partial* specifications of system behavior. They are admittedly incomplete, but useful nevertheless. Our seeding, and subsequent detection, of faults in container implementations illustrates their utility. A more thorough study of the kinds of faults that can be revealed by such specifications would provide a better understanding of the breadth of applicability of model checking for fault-detection.

In [12], we use abstract interpretations of the behavior of container implementations in checking properties of systems that use containers. These AIs are finite-state, thus it would be possible to encode them as an LTL formula and to check that property against container implementations. This would essentially lift model checking results for containers up to model checking of applications and provide a practical illustration of the reuse of verification results.

8 Conclusions

We have applied existing model checking tools to validation of the correctness of common container data structure implementations. In doing this, we have illustrated how crucial correctness properties of these systems can be encoded in LTL. We have applied a methodology that incorporates techniques from abstract interpretation and partial evaluation to construct finite-state models from the source code of container implementations. We believe that this study demonstrates the potential of model checking as a practical means of detecting faults in general purpose software components. This work raises a number of research issues whose study may further expand the breadth of applicability of model checking to common types of software systems.

Acknowledgements: The authors would like to thank James Corbett and George Avrunin for making the INCA toolset available to us, for enhancing its predicate definition capabilities, and for many valuable discussions about model generation.

References

1. J.-R. Abrial, E. Börger, and H. Langmaack. *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control.* Lecture Notes in Computer Science, 1165. Springer-Verlag, Oct. 1996.
2. G. Avrunin, U. Buy, J. Corbett, L. Dillon, and J. Wileden. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE Transactions on Software Engineering*, 17(11):1204–1222, Nov. 1991.
3. J. Beidler. The Scranton generic data structure suite. `http://academic.uofs.edu/faculty/beidler/ADA/default.html`, 1996.
4. E. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. Long, K. McMillan, and L. Ness. Verification of the future-bus+ cache coherence protocol. *Formal Methods in System Design*, 6(2), 1995.
5. J. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3), Mar. 1996.
6. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
7. D. Craigen, S. Gerhart, and T. Ralston. An international survey of industrial applications of formal methods. Technical report, National Institute of Standards and Technology, Mar. 1993.
8. J. Davies and J. Woodcock. *Using Z: Specification, Refinement and Proof.* Prentice Hall, 1996.

9. D. Dill, A. Drexler, A. Hu, and C. H. Yang. Protocol verfication as a hardware design aid. In *Proceedings of the IEEE International Conference on Computer D esign: VLSI in Computers and Processors*, pages 522–525, July 1992.
10. M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, May 1999. to appear.
11. M. Dwyer and D. Schmidt. Limiting state explosion with filter-based refinement. In *Proceedings of the 1st International Workshop on Verification, Abstract Interpretation and Model Checking*, Oct. 1997.
12. M. B. Dwyer and C. S. Păsăreanu. Filter-based model checking of partial systems. In *Proceedings of the Sixth ACM SIGSOFT Symposium on Foundations of Software Engineering*, Nov. 1998.
13. M. B. Dwyer, C. S. Păsăreanu, and J. C. Corbett. Translating ada programs for model checking : A tutorial. Technical Report 98-12, Kansas State University, Department of Computing and Information Sciences, 1998.
14. J. Hatcliff, M. B. Dwyer, and S. Laubach. Staging static analysis using abstraction-based program specialization. In *LNCS 1490.* Principles of Declarative Programming 10th International Symposium, PLILP'98, Sept. 1998.
15. D. Hoffman and R. Snodgrass. Trace specifications: Methodology and models. *IEEE Transactions on Software Engineering*, 14(9):1243–1252, Sept. 1988.
16. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.
17. C. Lewerentz and T. Lindner. *Formal Development of Reactive Systems: Case Study Production Cell.* Lecture Notes in Computer Science, 891. Springer-Verlag, Jan. 1995.
18. B. Liskov and J. V. Guttag. *Abstraction and Specification in Program Development.* The MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, MA, 1986.
19. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1991.
20. K. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.
21. G. Naumovich, L. Clarke, and L. Osterweil. Verification of communication protocols using data flow analysis. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Oct. 1996.
22. A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. Apt, editor, *Logics and Models of Concurrent Systems*, pages 123–144. Springer-Verlag, 1985.
23. P. Wolper. Specifying interesting properties of programs in propositional temporal logics. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*, pages 184–193, St. Petersburg, Fla., Jan. 1986.